Equivalence Proofs for Erlang Refactoring

Erica Tanti and Adrian Francalanza

Department of Computer Science, University of Malta
{erica.tanti.09 | adrian.francalanza}@um.edu.mt

Erlang [1, 2] is an actor-based programming language used extensively for building concurrent, reactive systems that are highlighly available and suffer minimum downtime. Such systems are often mission critical, making system *correctness* vital.

In industrial-scale systems, correctness is usually ascertained through *testing*, a lightweight verification technique trading analysis completeness for scalability. In such cases, a system is deemed *correct* whenever it "passes" a suite of tests, each checking for the correct functionality of a particular aspect of a system. This is also true for large Erlang systems: even when correctness specifications are provided, it is commonplace for Erlang developers to use *testing tools*, automating test-case generation from these specifications [6, 9].

Testing for concurrent systems is an arduous task. Since a concurrent system may have multiple execution paths—due to different thread interleavings—it is often the case that test runs fail to detect errors, only for them to crop up once the system is deployed. Because of this aspect, Erlang tests are often executed *multiple times* using tools that induce different interleavings [3], in the hope that enough execution paths are covered so as to be able to conclude (with a reasonable degree of certainty) that the error tested for is absent. To make matters worse, every time a system needs to be altered—because of either code maintenance, bug fixes or functionality enhancements—the entire testing proceedure needs to be carried out again from scratch.

In most cases, the a code update is often expected to pass the *same* test suite that the previous code had originally passed (possibly extended by additional tests). This is particularly the case for *refactoring*, code restructuring that does not necessarily change functionality, but instead makes the code more readable, compliant to certain code paractices, or more efficient. There are numerous tools assist or automate the refactoring process in Erlang systems [7, 8, 10]. However, none of these tools comes equipped with a guarantee that the substitute code preserves the behaviour of the code it is substituting, which can potentially violate correctness.

In this work, we strive towards a solution for this problem. We study a *testing* preorder [5, 4] for Erlang programs P, Q and Erlang tests T. We limit our study to *safety* testing suites, *i.e.,* suites of tests ensuring that *nothing bad happens*. Our safety testing preorder, $P \leq_{\text{safe}} Q$, denotes that P is *as safe as* Q, and is formally defined as:

$$P \leq^{\text{test}}_{\text{safe}} Q \qquad \text{iff} \qquad \text{for all } T \ \Big(\big(P \text{ fails } T\big) \ \text{ implies } \ \big(Q \text{ fails } T\big) \Big)$$

Note that, by the contrapositive, our testing preorder ensures that whenever Q passes a test T, P also passes that test.

This preorder may be used as the semantic basis for the aforementioned Erlang refactoring tools. More specifically, in a setting where correctness means passing a suite of safety tests, a refactoring tool would be considered *safe* whenever it substitute a program Q with another program P that can be shown to be as safe as Q, *i.e.,* $P \leq^{\text{test}}_{\text{safe}} Q$.

Unfortunately, reasoning about the preoder $\leq^{\text{test}}_{\text{safe}}$, even when limited to safety tests, is generally non-trivial because its definition relies on a *universal quantification* over all possible (safety) tests. We therefore investigate a theory that facilitates reasoning about our testing preorder. In particular, we develop an alternative trace-based preorder [5, 4] for Erlang programs; it relies on program traces s, instead of tests, and would normally take the following general format:

$$P \leq^{\text{trace}}_{\text{safe}} Q \qquad \text{iff} \qquad \text{For all } s \ \Big(\big(Q \text{ produces } s\big) \ \text{ implies } \ \big(P \text{ produces } s\big) \Big)$$

A trace based preorder is simpler to reason about because (1) interactions with a number of tests may be described using a single trace; (2) traces are somehow connected to the capabilities of the programs P, Q being analysed, whereas tests may not. The main result of our work would then be to prove a correspondence between the testing preorder, $\leq^{\text{test}}_{\text{safe}}$, and the trace based preorder, $\leq^{\text{trace}}_{\text{safe}}$.

However, the technical development does not follow directly from the work on testing preorders by Hennessy *et al.* [5, 4], and certain characteristics pertaining to actor systems complicate the development of our trace-based preorders. For instance, actors often interact through *asynchronous communication*, which reduces the tests' power of observation. Put differently, asynchrony increases the number of trace executions that may lead a test to fail, thus making the above trace-based preorder too rigid. Another characteristic is actor *persistence*, meaning that once an actor is spawned, it is receptive to an infinite number of messages; this yields an infinite number of traces even for the simplest of programs, making trace-based analysis unwieldy.

Through a series of examples, our talk will explain the problems encountered when developing our testing theory for Erlang programs and discuss the potential solutions being investigated.

References

1. Armstrong, J.: Programming Erlang - Software for a Concurrent World. The Pragmatic Bookshelf (2007)
2. Cesarini, F., Thompson, S.: ERLANG Programming. O'Reilly (2009)
3. Claessen, K., Palka, M., Smallbone, N., Hughes, J., Svensson, H., Arts, T., Wiger, U.: Finding race conditions in Erlang with Quickcheck and Pulse. In: ICFP. pp. 149–160. ACM, New York, NY, USA (2009)
4. De Nicola, R., Hennessy, M.C.B.: Testing equivalences for processes. Theoretical Computer Science 34(1-2), 83–133 (1984)

5. Hennessy, M.: Algebraic Theory of Processes. Foundations of Computing, MIT Press (1988)
6. Hughes, J.: Quickcheck testing for fun and profit. In: PADL '07. pp. 1–32. Springer-Verlag, Berlin, Heidelberg (2007)
7. Li, H., Thompson, S.: Testing Erlang Refactorings with QuickCheck. In: IFL '07. Freiburg, Germany (September 2007)
8. Lövei, L., Hoch, C., Köllö, H., Nagy, T., Nagyné Víg, A., Horpácsi, D., Kitlei, R., Király, R.: Refactoring module structure. In: ERLANG'08. pp. 83–89. ACM, New York, NY, USA (2008)
9. Papadakis, M., Sagonas, K.: A PropEr integration of types and function specifications with property-based testing. In: Erlang '11. pp. 39–50. ACM Press, New York, NY (Sep 2011)
10. Sagonas, K., Avgerinos, T.: Automatic refactoring of Erlang programs. In: PPDP '09. p. 13. ACM Press, New York, New York, USA (2009)